みが発生すると実行中のプロセスが一時中断されて，**割込み処理**が実行される．

割込み処理機能は**割込みハンドラ**（interrupt handler）ともいい，その処理の大半がOSの機能として実現される．割込みの発生による割込み処理の具体的な処理手順はおおよそ次のようになる．

(1) 割込みの受付

(2) 割込み要因の識別

(3) プロセスが共用するハードウェア資源の内容の退避

(4) 割込み要因ごとの割込み処理

(5) 退避していたハードウェア資源の内容の回復

(6) 割込み処理の完了

割込み処理中は原則として割込み禁止とし，他の割込みが発生しても現在の割込み処理が終了するまで，新たな割込みの受付けは待たされる．この原則が覆る場合や複数の割込みが同様のタイミングで生じた場合の処理方式については，〔2〕で詳述する．

また，割込み処理手順は，実際には，制御アーキテクチャとしてハードウェア機構（マイクロプログラムを含む）とOSを核とするソフトウェアの機能分担によって実現され，トレードオフがある．たとえば，(1)はソフトウェアでは実現不可能なのでハードウェア機構が，(2)は高速処理を要求されるのでハードウェア機構が，(3)(5)(6)はハードウェア機構とユーザプログラムとの橋渡しであるのでOSが，(4)は割込み要因ごとに処理内容が異なるのでOSを核とするソフトウェアが，それぞれ分担する方法が考えられる．

〔2〕 **多重レベル割込み** 2.3.1項〔2〕で述べたように，割込みにはその要因によって多種多様なものがある．また，これらは独立した事象であるものが大半であり，同時に発生する可能性もある．プロセッサを1つしか持たないノイマン型コンピュータにおいては，複数の割込みについても逐次的に処理せざるをえない．そのための機構が**多重レベル割込み**（優先度付き割込み）機構である．多重レベル割込み機構では，割込み要因ごとに割込み処理の優先度を付与し，その高低（緊急性）によって複数の割込みを順序付けて処理する．すなわち，多重レベル割込み機構は

- ある割込み処理時には，それよりも優先度の低い割込みは禁止される
- 一方，ある割込み処理時にそれよりも優先度の高い割込みが発生した場合

には，後者が優先処理される（割り込まれる）

というようにして複数割込みの逐次処理を制御する．

優先度は割込み処理の緊急性（必要性）によって順位付けする．たとえば，優先度が高い要因から順に，① 緊急処理が必要なハードウェア的割込み；② ソフトウェア的割込みのうちで命令実行例外やメモリアクセス例外などのマシン命令の実行に同期する割込み；③ 迅速な処理が必要な（高速入出力装置からの）入出力割込み；④ 低速入出力装置からの入出力割込み；⑤ ユーザプロセスなどにおいて，マシン命令の実行が起こす内部割込み；などとする．ただし，これは例であり，優先度の付与方法についてはコンピュータアーキテクチャごとに種々ある．

〔3〕 **割込みのタイミング** 割込み処理には割込み処理前の状態（**マシン状態**という，ハードウェア資源の内容など）の退避と割込み処理後のそれの回復が必要であり，中途半端なタイミングで本来の処理を中断すると退避すべきマシン状態が増えたり，割込み前に復帰できないなどのオーバヘッドが生じるおそれがある．このため，割込みは原則としてマシン命令とマシン命令との間で受け付けられる．

したがって，割込み要因の発生から実際に割込みが受け付けられて割込み処理に入るまでには割込み受付けのタイミングによって待ち時間が生じる．これを**割込み待ち時間**という．

緊急性の高い高優先度の割込み待ち時間が長くなると種々の問題が生じるので，次のような対策が考えられている．

- 緊急を要する超高優先度の要因は即時に受け付ける．
- 実行時間の長い命令は実行途中でも割込みを許可する．
- 中断した命令の割込み処理後の再実行（再開）を可能とし，割込みを受け付ける．
- プログラム（OS）で割込み許可時間（期間）を制御する．

〔4〕 **割込み受付けタイミングと割込み処理からの復帰方法** また，割込み処理後の措置すなわち割り込まれたマシン状態の回復方法は，前の〔3〕で述べた割込みの受付けタイミングおよびその割込みを生じさせた要因と密接な関係がある．たとえば，次のような受付けタイミングと割込み処理後の措置の方法がある．

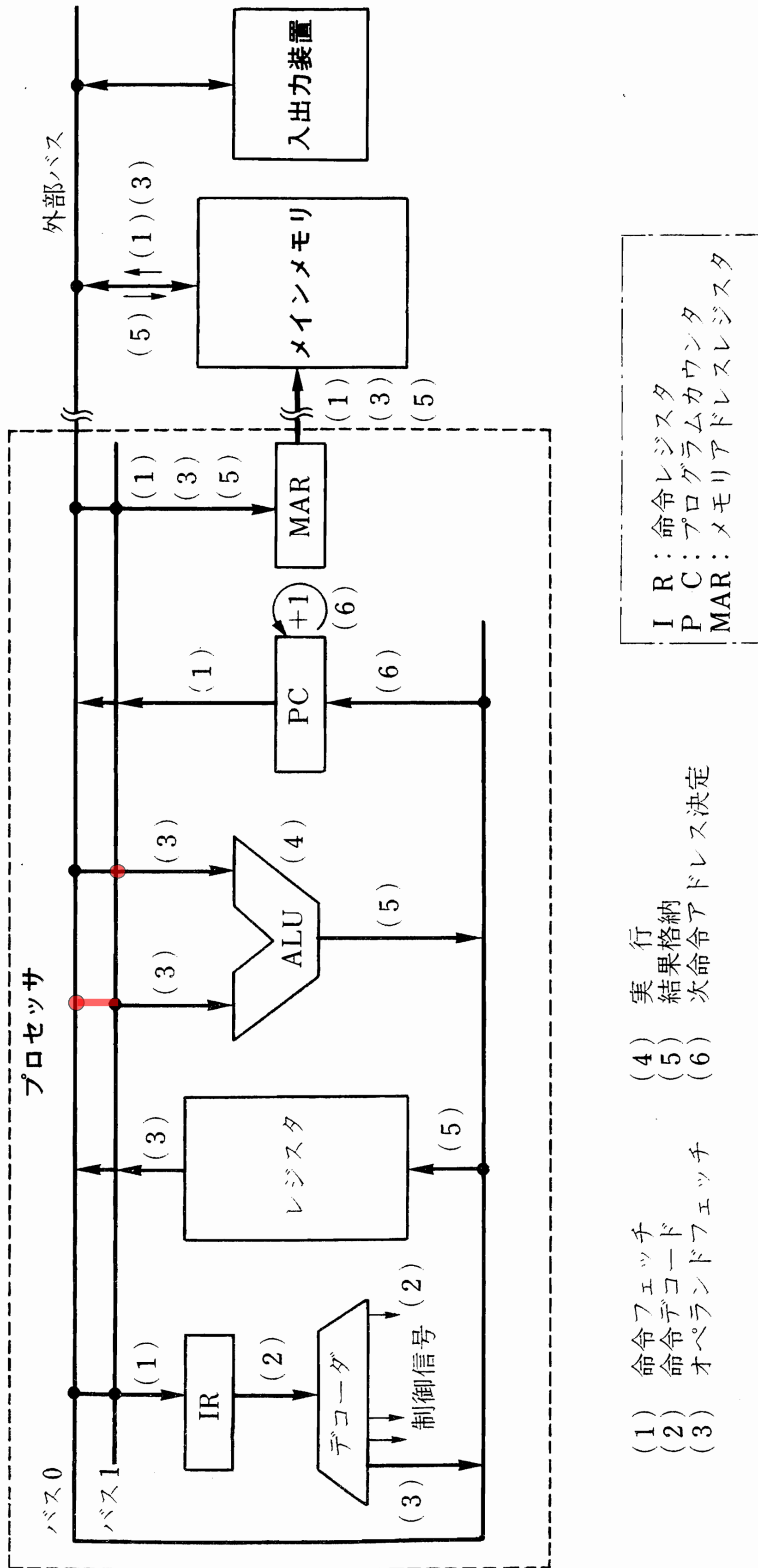

図 2.25 ハードウェア構成例と命令実行サイクル

2.3 割 込 み

本節では，時間的/空間的多重化によるハードウェア資源の共用を駆使する現代のノイマン型コンピュータでは必須機能である**割込み**（interrupt）について述べる．

2.3.1 割込みとは？

〔1〕 **割込みの必要性** 割込み機能は本来の（プログラムとして明示されている）命令実行順序を強制的にかつ動的に変える手段である．

現代のノイマン型コンピュータに割込み機能が必須である理由は

① プログラムに記述していないあるいは記述できない不測の事態に対処する

② ハードウェアやプログラムの本来の機能以外の動作，すなわち異常/エラー/例外などを検知し，それに対処する

③ ハードウェア機構あるいはユーザプログラム（ここでは，OS以外のプログラム）と基本ソフトウェア（特に，OS）との通信手段を与える

④ 共用ハードウェア資源（プロセッサやメモリなど）の利用要求の競合を解決（スケジューリング）し，それらの効率的な利用を図る

⑤ 互いに非同期動作している資源（たとえば，プロセッサと入出力装置，ネットワークでつながれたプロセッサ相互など）を通信などのために，あるタイミングで同期させる

の各機能を実現するためである．

割込みが生じると，本来の（現在実行しているプログラムの）実行順序とは別の処理（**割込み処理**という）へ制御フローが分岐する．

〔2〕 **割込み要因** 割込みには必ずそれを引き起こした要因（原因，事象）がある．これを**割込み要因**という．割込み要因を分類する指標として，図2.26に示すように

(1) **内部か外部か**？：要因の発生個所がコンピュータの内部（プロセッサかメモリ）なのか，外部（入出力装置）なのか？

(2) **ハードウェア的かソフトウェア的か**？：(1)の内部要因の細分類指標で

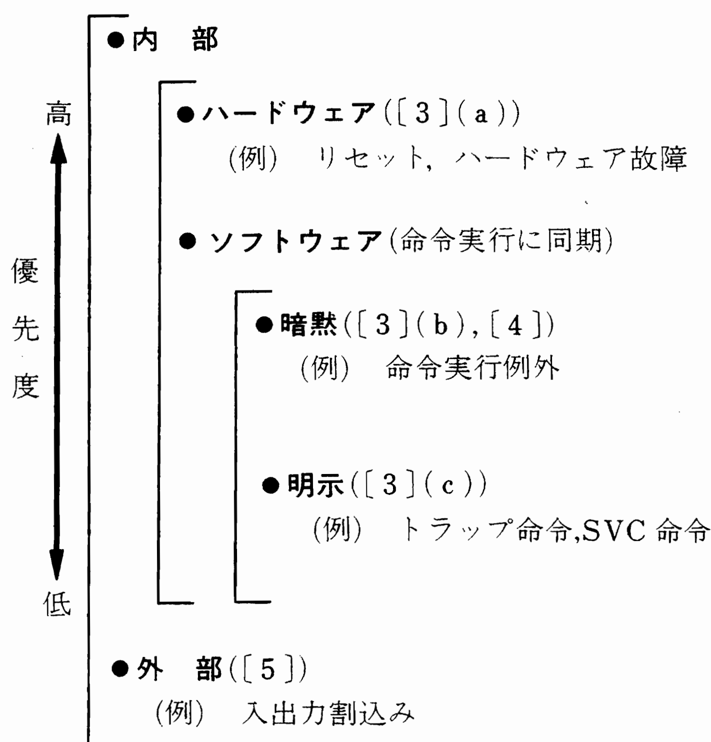

図 2.26　割込み要因の分類

あり，ハードウェア装置が要因となるものか(ハードウェア的)，プログラム（マシン命令）の実行などのソフトウェア的要因によるものか？　ハードウェア的要因は「命令実行とはまったく関係のないことが原因である(命令実行と非同期)」，ソフトウェア的要因は「あるマシン命令を実行したことが原因である（命令実行に同期)」といえる．

(3)　**マシン命令として明示してある（明示的）かそうでない（暗黙的）か？**：(2)のソフトウェア的（命令実行に同期する）要因の細分類指標であり，そのマシン命令を実行すると割込みが生じることが命令機能として明示されているか（明示的)，明示されていないか（暗黙的)？

などがある．

〔3〕　**内部割込み**　割込み要因の発生個所がコンピュータ内部（プロセッサかメモリ）である割込みであり，**割出し**あるいは**トラップ**(trap) ということもある．

〔2〕の(2)～(3)で示した分類指標に従って，内部割込み要因の代表例について分類してみよう．

(a)　**ハードウェア的割込み要因**：リセットスイッチ（ボタン）がオンされたことによるリセット(reset)，あるいは電源異常やパリティエラ

(a) **命令完了**：実行中のマシン命令の完了まで割込みの受付けを待たせる．(例) 入出力割込み．

(b) **命令抑制**：実行中の命令を NOP (no operation：無操作) 命令にし，その NOP 命令を実行後割込みを受け付ける．(例) アクセス権違反 (アクセス禁止領域への侵犯)．

(c) **命令無効**：実行中の命令を中断する．プログラムカウンタは不変とするので命令は未実行扱いとなり，再実行 (再開) 可能 (〔5〕参照) である．再実行するかどうかは割込み要因によって，OS が判断する．(例) ページフォールト．

(d) **命令中止**：実行中の命令を即時に中止し，(途中でも) 強制終了してしまう．中止された命令は再実行されないのが普通である．(例) ハードウェア故障，リセット．

(a)(b) はマシン命令とマシン命令との間に割込み処理を挿入しているので，ハードウェアレベルでの資源の排他制御は不要である．

これらに対して (c) は割り込まれたマシン命令が中断し未完了であるので，その命令を無効とするには途中まで行った処理を元に戻す処理 (**アンドゥ*** という) が必要となる．現代のコンピュータのようにハードウェア資源の複雑な多重化を行っている場合のアンドゥ処理は複雑である (5.2.1 項〔10〕参照)．

(d) はリセットと同義であり，そのマシン命令の処理結果は保証されないだけではなく，アンドゥ処理も行うことはできない．

実際のコンピュータは OS の管理下で動作しており，割込み処理は OS 機能のコア部分として実行される．したがって，(a)〜(d) のいずれの場合にも，原則として，OS の管理下で割込み処理や割込み処理後の措置は実行される．

〔5〕 **正確な割込み** 〔4〕で述べたように，割込みは種々のタイミングで生じるので，割込み処理では，割り込まれたマシン命令に対する措置も重要となる．特に，(c) の「命令無効」では

(1) **回復** (recovery)：割り込まれた (無効となった，中断した) 命令の影響を取り除く (アンドゥ)

(2) **再開** (restart)：割り込まれた命令を再実行する

* アンドゥ (undo)：(マシン命令の) 実行によるレジスタの書換えなどの影響を消し，その命令を未実行あるいは実行前のマシン状態に戻すこと．

用コンピュータによる汎用プログラミング言語の処理は現代のノイマン型コンピュータにおける最も典型的なハードウェア/ソフトウェアトレードオフといえる.

汎用コンピュータと専用コンピュータの比較については，3.2.4 項でも述べている.

〔4〕 **コンパイラの機能とコンパイル指向プログラミング言語** プログラミング言語とコンピュータアーキテクチャとのセマンティックギャップを埋める機能（言語処理）は，普通**コンパイラ**というソフトウェアが分担する．すなわち，図 4.14 に示したように，プログラミング言語で記述されたプログラムを，実行に先立ってコンパイラによってマシン命令列に**コンパイル**するのである.

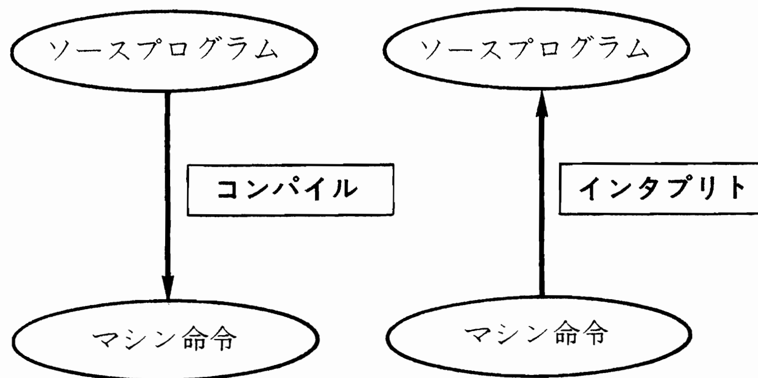

図 4.14 コンパイルとインタプリト

図 4.13 のたとえでは，コンパイラはプログラミング言語から対象コンピュータのマシン命令（列）へ渡るための渡し船であり，対象とする高級プログラミング言語やマシン命令セットが変わればコンパイラを書き直せばよい.

主としてコンパイラによって処理されるプログラミング言語は**コンパイル指向プログラミング言語**と呼ばれる．コンパイルというプログラミング言語処理では，プログラムの実行前に（静的に）実行可能な最適化処理をできる限り行い，実行時に余計な（言語）処理を持ち越さないようにして，ハードウェアによるマシン命令実行の最適化（高速化）を図っている．Fortran, Cobol, C, Ada, PL/I, Algol などの手続き型プログラミング言語の大半は，コンパイル指向プログラミング言語といえる.

しかし，プログラミング言語処理では，プログラムの実行のみではなく，プログラムの開発過程（プログラミング）におけるオーバヘッドを少なくするこ

語機能を活用することができる.

④ プログラミング言語処理に必要なハードウェア機能が明確になる.

⑤ 動的なデータ型検査を必要とするプログラミング言語処理に向いている.

Lisp や Java などの会話型プログラミング言語や OS のコマンドやシェル* 言語はプログラムの開発環境にも配慮して設計されており，これらは**インタプリト指向言語**と呼ばれる.

〔6〕 **コンパイル-インタプリト** 概念的には，セマンティックギャップをプログラミング言語側から埋めていくのがコンパイルであり，逆に，マシン命令側から埋めていくのがインタプリトである．さらに，それぞれのトレードオフを考慮して，セマンティックギャップにおける適当な機能レベルを**中間言語**レベルとして定め，コンパイラとインタプリタとを併用してギャップを埋めようとする言語処理方式を**コンパイル-インタプリト**あるいは**中間言語インタプリト**という．この場合の中間言語の機能レベルは仮想的なコンピュータのマシン命令と考えることができ，この中間言語のインタプリトを**仮想マシン**(4.2.6 項〔4〕で詳述）ということもある．なお，中間言語は，高級プログラミング言語と対象マシン命令とのセマンティックギャップが大きい場合のコンパイルにおいて，言語処理の途中段階の機能レベルとしても設けられ利用される（〔4〕参照）.

図 4.15 に示すように，コンパイル-インタプリトによるプログラミング言語処理では，① 高級プログラミング言語プログラムをいったん中間言語プログラムにコンパイルする；② その中間言語プログラムを（マシン命令プログラムによって）インタプリトする；の 2 段階でプログラミング言語処理を行う．中間言語のインタプリタがマシン命令プログラムではなくマイクロプログラムで記述されている場合は中間言語レベルがマシン命

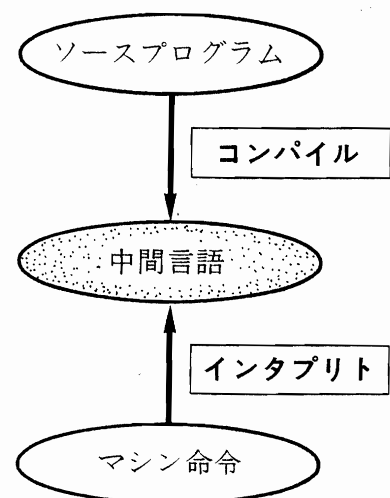

図 4.15 コンパイル-インタプリト

* シェル (shell)：ユーザインタフェースを実現する OS のコマンド処理系.

令レベルとなる（このマイクロプログラムによるマシン命令や中間言語のインタプリトを**エミュレーション**という，4.5節で詳述）．これは間接実行型高級言語マシン（4.2.6項〔4〕で詳述）の一種である．

コンパイルあるいはインタプリトの利点を活用するために，両者を場合に応じて使い分けることが可能なプログラミング言語処理系もある．たとえば，Lisp, Prolog, Javaなどのインタプリタによる会話型処理を基本とするがコンパイル処理も可能なプログラミング言語である．

また，中間言語をあるマシン命令セットとしてコンパイラの出力対象とし，実行時にそのマシン命令を別のマシン命令によってインタプリト（エミュレーション）する，あるいは別のマシン命令に変換する，という方式もある．たとえば，図4.16に示すように，あるCISC（マイクロプロセッサ）の互換チップ(互換性のあるマイクロプロセッサチップ)をRISCアーキテクチャによって実現する実例がある．この例では，コンパイラが生成したCISC命令列を実行時にRISC命令列に変換・実行する．（CISC，RISC，互換性については7.1節で詳述）

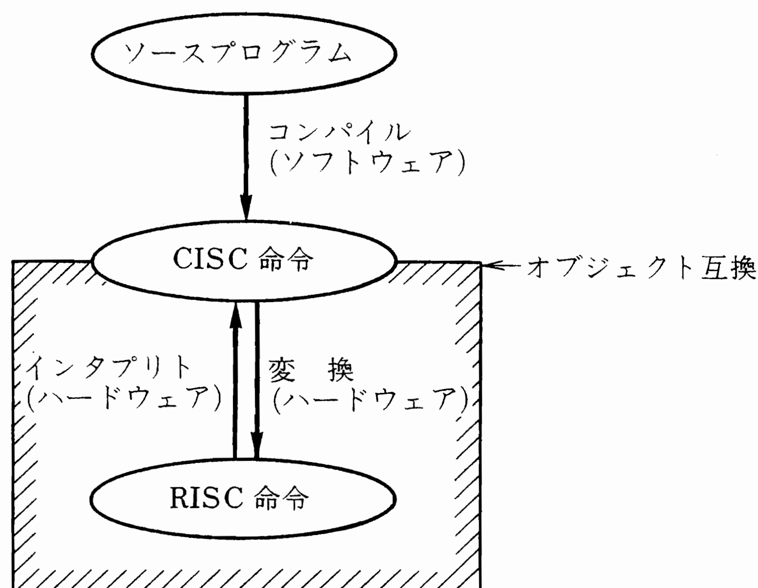

図 4.16 CISC命令を中間言語とするプログラミング言語処理例

〔7〕 **実行時コンパイル** 高級プログラミング言語プログラムをそのまま，あるいはそれに近い機能レベルの中間言語プログラムを，実行時に必要部分だけコンパイルしながら実行するプログラミング言語処理方式である．実行時ではあるがマシン

ン命令によっても左右される.

すなわち,「コンピュータアーキテクチャあるいは命令セットアーキテクチャの決定」とは「コンパイル機能におけるハードウェアとコンパイラとの機能分担方式の決定」を意味する.

〔2〕 **コンパイラ支援アーキテクチャ** システムプログラム(ソフトウェア)としてのコンパイラの機能はプログラミング言語やマシン命令の機能レベルによって決まるが,特にハードウェア/ソフトウェアトレードオフとしてのコンピュータアーキテクチャはマシン命令セットであるので,アーキテクチャ設計においてはハードウェアとコンパイラとのトレードオフが重要となる.

対象コンピュータ(ハードウェア)が決定されている場合のコンパイラ設計では,ハードウェアとコンパイラとのトレードオフは問題にならない.しかし,コンピュータアーキテクチャの設計においては,ハードウェア/ソフトウェアトレードオフの一端を担うコンパイル機能におけるハードウェアとコンパイラとの機能分担方式を決めなければならない.

したがって

(a) コンパイル機能を支援するハードウェア機構

(b) ハードウェア機構による実行を支援するコンパイラ機能

とが考えられ,これらを**コンパイラ支援アーキテクチャ**という.

図4.13のたとえでは,(a)はハードウェアによる分担機能を大きくし,コンパイル機能を支援しようとするもので,一方,(b)はコンパイラ(ソフトウェア)による分担機能を大きくし,ハードウェア(マシン命令)の実行を支援しようとするものである.(b)はコンパイラによる**コード最適化**(code optimization)ともいわれる.

〔3〕 **コンパイラ支援機能** 前の〔2〕で述べた(a)(b)はトレードオフとなっており,アーキテクチャ設計にはそれぞれを個別に考えるのではなく,併せてこのトレードオフを定めなければならない.

コンパイラ支援アーキテクチャとして汎用性のある具体的な機能として次のようなものが考えられる.

(a) **実行支援**:**コードスケジューリング**(4.3.2項参照),**分岐予測**(5.4.4項参照),プリフェッチやキャッシュ制御用マシン命令の挿入など.

(b) **プログラミング支援**:デバッグ支援(〔4〕参照)など.

OS機能であり，また，OSには必須の機能である．

同期機能の，プロセス間通信以外のOS機能への応用としては

- **動的負荷分散**：分散処理システムや並列コンピュータにおいて，複数プロセッサの負荷をできるだけ均等にするようにOSが動的にプロセス割当てを変更する機能
- **バリア同期**（barrier）：図4.26に示すように，複数プロセスすべての終了による同期機能
- **マルチプロセッサ制御**（7.2.3項〔21〕参照）

などがある．

4.4.4 マルチスレッド化

〔1〕 **プロセスとスレッド** プロセスが生成した命令実行（制御）の流れ（命令ストリーム，制御フロー）を**スレッド**（thread）という．プロセスはメインメモリに割り付けられた実行主体の最小単位であり，ハードウェア資源（特にメインメモリ）への命令やデータの割付け単位を意味する．これに対して，スレッドはプロセッサにおいてプロセスが生成した動的な実行単位（命令ストリーム）である．したがって，1つのプロセス中には少なくとも1つのスレッドが存在し，プロセス中の複数スレッドはプロセスに割り付けられた資源を共有する．また，プロセッサがスレッドを実行する際にはそれが含まれるプロセスがメインメモリ上に割り付けられていなければならない．

図4.27に示すように，スレッドは，具体的には，ループや基本ブロック* などの制御フローであり，プロセスに割り付けられた資源や環境（コンテキスト）よりもずっと有効時間が短く局所的な実行環境（**スレッドコンテキスト**（thread context）という）を持つ．典型的なスレッドコンテキストとしてレジスタ（の一部）がある．したがって，**スレッドスイッチ**（thread switch）はプロセススイッチよりもずっと軽い操作である．

コンテキストを必要最小限に限ったプロセスを**軽量プロセス**（lightweight process）といい，一般的には，「軽量プロセス」は「スレッド」と同義とされる．

また，ユーザが書いたプログラムとプロセスは簡単に対応付けられるが，プ

---

* 基本ブロック（basic block）：そのコード列中に分岐命令および分岐先を持たない命令ブロック．コードスケジューリングの基本単位として用いる．

前提に，コンピュータアーキテクチャの主要部分を構成している．

(a) プロセッサと同じデバイス（IC）技術で実現される **IC メモリ**である．

(b) 格納（記憶）のほかに，**読出し**（read）と**書込み**（write）の機能を持つ．

(c) 命令とデータを格納する（**プログラム内蔵**方式の実現）．

(d) **ランダムアクセス性**：アドレス指定だけで任意の格納場所を一意に識別して，一定時間でそこにアクセスできる．この性質を備えたメモリを**ランダムアクセスメモリ**（random access memory：**RAM**）という．

(e) **線形アドレス性**：1 次元アドレスによって格納場所（アクセス先）を指定できる．

(d) および (e) がメモリアーキテクチャ設計に及ぼす影響については〔2〕で述べる．

〔2〕 **メインメモリとメモリアーキテクチャ**　メインメモリは，「アドレスを指定しさえすれば任意の場所に一定のアクセス時間でアクセスできる」ランダムアクセス性と，「アドレスをインクリメント（“1” ずつ増加）するだけで並び順のアドレスに順番にアクセスできる」という線形アドレス性とを特徴とする．この 2 つの特徴の両立がノイマン型コンピュータのメインメモリの要件である．

すなわち，このメインメモリのランダムアクセス性と線形アドレス性は

- ハードウェア構造が単純な繰返し構造となり，ある機能レベルでの容量とアクセス速度の両立をトレードオフポイント（〔4〕で述べるメモリ階層）として実現できる
- プログラムにおけるアクセスパターンによってランダムアクセス性と線形アドレス性とを使い分けることが可能となる

というメモリアーキテクチャの設計における要点を生み出している．

〔3〕 **メモリの性能**　メモリの性能を定量的に評価する指標として

(a) **容量**：格納できる情報の最大数（単位はバイト）であり，メモリ媒体の記録密度（単位面積当りの格納可能ビット数）などによって左右される空間的性能指標である

(b) **アクセス速度**：メモリへのアクセス要求を発してから実際にメモリ内のデータにアクセス（読出し/書込み）できるまでに要する時間（**メモリアクセス時間**という，単位は秒）あるいはメモリが連続して要求を受付け可能な最小時間

間隔（**メモリサイクル時間**という）で表す時間的性能指標であるとがある．

厳密にいうと，メモリサイクル時間 T_{MC} は，①（厳密な意味での）メモリアクセス時間 T_{MA}：プロセッサからのメモリアクセス要求やアドレスがメモリに伝えられるのに要する時間；② 読出し/書込み時間 T_{RW}；③ その他の付随的な処理時間 T_O：たとえば，DRAM におけるプリチャージ時間（6.4.4 項〔3〕参照）；の合計となる．

$$T_{MC} = T_{MA} + T_{RW} + T_O \tag{6.1}$$

メモリアクセス時間 T_{MA} の短縮はメモリスループット（6.4.2 項〔1〕で詳述）の向上に直接結びつく．また，それによるメモリサイクル時間 T_{MC} の短縮はメモリアクセス競合時の待ち時間を短縮させる．

〔4〕 **メモリ階層**　メモリ性能の指標である容量とアクセス速度とはトレードオフとなる．この容量とアクセス速度とのトレードオフがメモリ機能を特徴付ける．すなわち，図 6.1 に示すように，容量（空間的性能指標）とアクセス速度（時間的性能指標）とのトレードオフによってメモリ機能の大きさが決まるので，その機能を実現するために適用するメモリ媒体などのデバイス技術が分類できる．容量とアクセス速度とのトレードオフによってできるメモリ機能レベルの種々を**メモリ階層**（memory hierarchy）という．

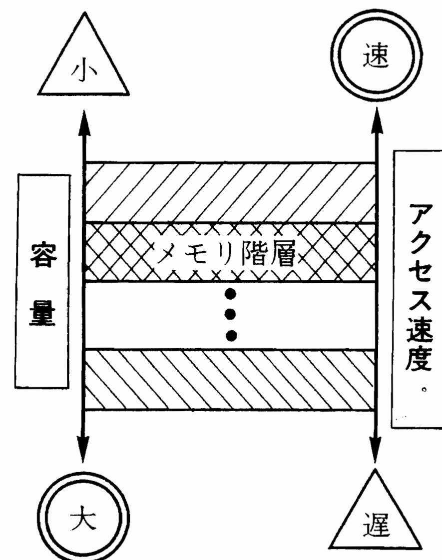

図 6.1　メモリ性能とメモリ階層

メモリ階層は，1.1.3 項〔4〕で述べたアーキテクチャ設計における対立概念のうち「空間 ⇔ 時間」に基づいてできる機能分類の代表例である．

〔5〕 **参照局所性**　ある一定の実行時間内では，プロセッサによってアクセス（参照）されるメモリのアドレス（格納場所）は一部に集中する．これを**参照局所性**（referential locality）あるいは「参照局所性が高い」という．

参照局所性は，局所性を示す対象によって

(a) **空間的参照局所性**：一度アクセスされたアドレスに近接する（格納場所が近い）アドレスは近いうちにアクセスされる可能性が高い

(b) **時間的参照局所性**：一度アクセスされたアドレスは近いうちに何回もアクセスされる可能性が高い

とに分けることができる.

命令（プログラム）とデータ，また適用する問題ごとに，参照局所性の傾向は異なる.また,参照局所性の性質はメモリアーキテクチャの設計における種々のトレードオフ決定に影響を及ぼす.

〔6〕 **メモリアーキテクチャ設計の要点**　メモリアーキテクチャ設計はメモリ機能におけるハードウェア/ソフトウェアトレードオフ設計である.したがって

- プロセッサとの関係（プロセッサアーキテクチャがメモリアーキテクチャに何を要求するか？），特にフォンノイマンボトルネック（2.1.2 項〔2〕参照）の解消
- メモリ階層の適用と利用
- 参照局所性の利用

を実現する方法を種々のトレードオフに基づいてメモリアーキテクチャとして盛り込むことがメモリアーキテクチャ設計の要点となる.

実際には，次のようなメモリアーキテクチャ設計の目標がある.

(a) **大容量化**：メモリの空間的性能の向上を図る．たとえば，メモリ階層を利用する仮想メモリ（6.2 節参照）がある.

(b) **高速化**：メモリの時間的性能の向上を図る．メモリそのもののアクセス速度の向上のほかに，プロセッサ ⇔ メモリ間の転送性能(メモリスループット，6.4.2 項参照)の向上などが考えられる．たとえば，① キャッシュ：メモリ階層の利用（6.3 節参照）；② メモリインタリーブ：プロセッサ ⇔ メモリ間スループットの向上に参照局所性を利用（6.4.2 項〔3〕参照）；③ 高速メモリインタフェース：プロセッサ ⇔ メモリ間スループットの向上（6.4.4 項参照）；④ ロード/ストアパイプライン：プロセッサアーキテクチャの対メモリ機能の改善（5.4.5 項〔1〕参照）；などである.

(c) **高機能化**：格納とアクセスのほかに機能を付加する．たとえば，タグメモリ（6.4.3 項〔5〕参照）などがある.

〔7〕 **メモリ機能におけるハードウェア/OS/コンパイラのトレードオフ**

メモリアーキテクチャは「プロセッサ(マシン命令)から見えるメモリの機能」とも,「ユー

ザ　　から見えるコンピュータのメモリの機能」ともいえる．いずれも「マシン命令で指定（利用）できるメモリ機能」と言い換えることができる．

メモリ機能の実現におけるハードウェアとソフトウェア(OS/コンパイラ，システムプログラム）の機能分担の代表例としては

(a) **ハードウェア機構**：たとえば，① メモリ（素子）やメモリインタフェースの高速化；② 仮想メモリの支援（DAT など，6.2 節参照）；③ キャッシュ機構の大半（6.3 節参照）；④ アドレス修飾

(b) **OS**：たとえば，① メモリ保護（メインメモリ保護やファイル保護）；② 仮想メモリ管理（アドレス変換やブロック置換，6.2 節参照）

(c) **コンパイラ**：たとえば，① 仮想メモリの利用（仮想アドレスによるコード生成）；② アドレス修飾（アドレッシングモード）方式の決定；③ 参照局所性を利用したコードスケジューリング；④ データ型によるメモリ割付け

などである．

6.1.2 メモリ階層とメモリアーキテクチャ

〔1〕 **ノイマン型コンピュータの主要なメモリ階層**　現代のノイマン型コンピュータにおいて，6.1.1 項〔3〕で述べた性能指標によって設定されるメモリ階層としては次の 3 階層が代表的である．（図 6.2 参照）

(a) **メインメモリ**：命令とデータの短期（(b) のファイル装置と比べると）格納用であり，文字通り主要なメモリである．プロセッサとともにコンピュータの内部装置を構成するので，**内部メモリ**，**1 次メモリ**あるいは単に**メモリ**ともいう．使用中の命令やデータを格納し，プロセッサが直接アクセスする．容量とアクセス速度の両性能とも

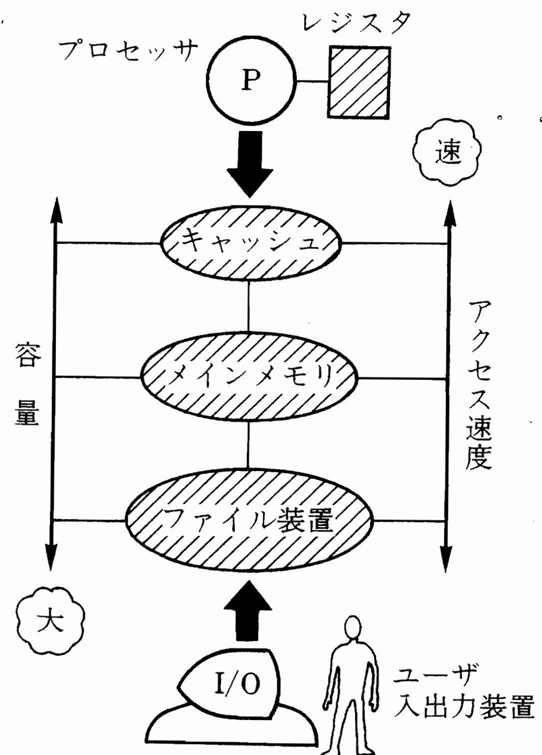

図 6.2 ノイマン型コンピュータの代表的メモリ階層

⑾ **大容量補助メモリ**：主要なファイル装置であるハードディスクのバックアップメモリとして位置付けられる．たとえば，光磁気/光ディスク（MO, CD-ROM など），磁気テープ（8mm テープ，DAT など）などがある．

以上の⑻〜⑾は電気/磁気/機械のそれぞれによる動作部分があり，電気的速度では動作しないが，不揮発性メモリである．

特に，近年のプロセッサ（レジスタ）の動作速度の向上は著しく，⑴のレジスタとICメモリではない⑻以降の補助メモリとの速度差は拡大する一方である．このため，① メインメモリの一部をディスクキャッシュとして利用する；② キャッシュの容量を拡大する；など，多様なメモリ階層の拡張利用が行われている．

〔4〕 **メモリ階層を利用したメモリアーキテクチャ**　メモリ階層を利用するメモリアーキテクチャとして次の2方式が代表的である．両者はいずれもメモリ階層を利用したメインメモリアーキテクチャの改善手法であるが，図6.3に示すように，改善目的やメモリ階層の利用方法，それらに起因するハードウェア/ソフトウェアトレードオフが相異なる．

● **仮想メモリ**：ユーザ（実際には，マシン命令生成者やマシン命令）に対して，メインメモリ容量

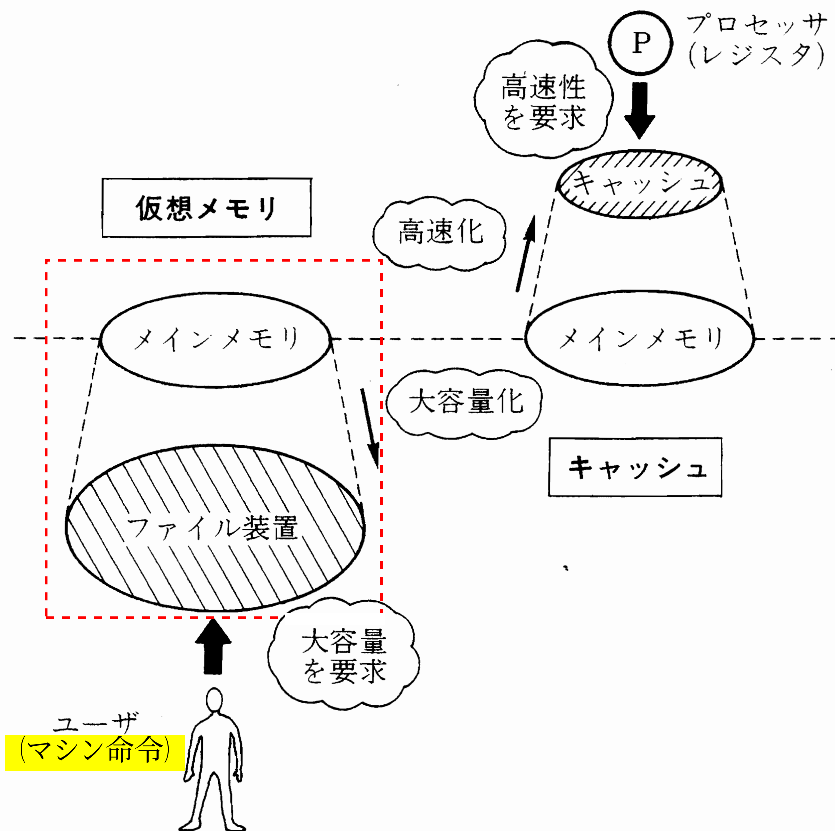

図 6.3　メモリ階層を利用するメモリアーキテクチャの改善手法

を仮想的に拡大して見せる方式である．ユーザ
のためのメインメモリ性能の空間的改善を目標としている点で，その実現においてはソフトウェア機能が重視される．(6.2節で詳述)

- **キャッシュ**：プロセッサに対して，メインメモリのアクセス速度を仮想的に高速化させて見せる方式である．仮想メモリとは逆に，プロセッサのためのメインメモリ性能の時間的改善を目標としている点で，その実現においてはハードウェア機能が重視される．(6.3節で詳述)

6.2 仮想メモリ

本節では，ユーザ（人間）に対してメインメモリ容量の見かけ上の拡大を図るメモリアーキテクチャである**仮想メモリ**(virtual memory)について述べる．

6.2.1 仮想メモリとは？

〔1〕 **仮想メモリの意義** 仮想メモリでは，メインメモリ（現代のノイマン型コンピュータではICメモリで実装）とファイル装置（補助メモリ，現代のノイマン型コンピュータではハードディスクで実装されるのが一般的）の2つのメモリ階層を利用する．

すなわち，図6.4に示すように，メインメモリと比べるとアクセス速度は遅いが大容量のファイル装置をメインメモリのバックアップメモリとする機能によって，ユーザ（実際には，コンパイラなどのマシン命令生成者，あるいはマシン命令そのもの）に対してメインメモリの容量を仮想的に大きく見せる方式である．マシン命令はオペランド対象としてファイル装置を意識しなくとも，メインメモリの容量の制限が緩やかになったり，なくなったように見える．仮想メモリは「実際のメインメモリの空間的性能を隠蔽して，その性能が改善されたようにユーザに見せかけるメモリアーキテクチャ」といえる．

仮想メモリは，1960年前後の第2世代のコンピュータシステムで実現されて以来，メモリ素子技術の変遷/進歩やメモリ階層の多様化にかかわらず，現代のノイマン型コンピュータのメモリアーキテクチャを支える重要な要素技術として定着している．

〔2〕 **仮想メモリと実メモリ** 仮想メモリ方式では，図6.4に示すように，

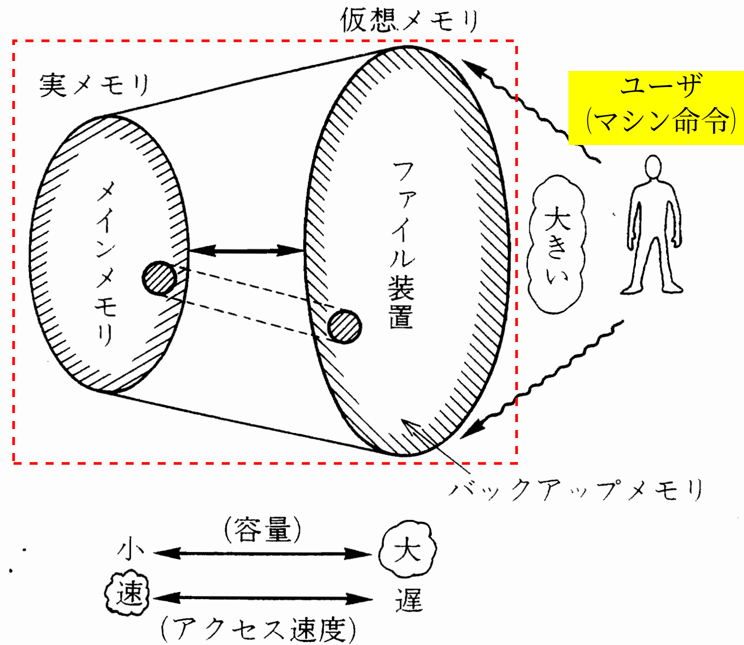

図 6.4　仮想メモリ

メインメモリ内の対象データのアドレスに関して

- マシン命令（プログラム）によって指定する**論理アドレス**（**仮想アドレス**）
- 実際のメインメモリ素子に付けられている**物理アドレス**（**実アドレス**）

をおのおの独立させる．そして，ユーザ(プログラム)やコンパイラ(マシン命令生成者)には，メインメモリ空間として論理アドレス空間(ファイル装置容量の空間規模がある)を仮想的に見せる．この論理アドレス空間を構成するメモリ（実体はファイル装置）を**仮想メモリ**という．これに対して，プロセッサなどのハードウェア機構が直接アクセスする物理アドレス空間である実際のメインメモリを**実メモリ**（real memory）という．

仮想メモリ機構は，論理アドレス空間（仮想アドレス空間）と物理アドレス空間（実アドレス空間）の対応関係（**アドレスマッピング**（address mapping）という）を管理/制御する機能を担う．

〔3〕 **仮想メモリの効果**　仮想メモリはハードウェア/ソフトウェアトレードオフに対して次のような効果を与える．

(a) **実アドレス空間サイズの制約を事実上なくしている．**

- コンパイラは，メインメモリの物理的な実装容量（実アドレス空間サイズ）による制約を受けることなく，マシン命令中に論理的なオペランドアドレス（仮想アドレス）を置くことができる．

- OSは，プログラムやデータを格納する場所の物理的なサイズの限界を考慮しなくても，それらを論理的意味のあるファイルのまま管理することができる．
- メインメモリの物理的な実装容量の変更が命令セットアーキテクチャに影響を及ぼさなくなるので，プログラムの良好な移植性やオブジェクト互換性の保持が可能となる．

(b) **実アドレス空間の効率的利用が可能となる．**

- コンパイラは，生成するマシン命令のオペランドとして仮想アドレスを使用するので，広い仮想アドレス空間を指定できる．
- OSは，広い仮想アドレス空間上で連続した領域をプロセスに割り付けることが可能となり，その単位で実アドレス空間にマッピングすれば，実アドレス空間のフラグメンテーション* を防止できる．

すなわち，仮想メモリは「メインメモリのハードウェアによる空間的制約をシステムプログラム（コンパイラやOS）から隠蔽する」という効果をもたらす．一方で，その効果を実現するために，仮想メモリ機能の効率的な運用ができるようにハードウェア/OS/コンパイラトレードオフを設定する必要がある．

6.2.2 仮想メモリ機構

〔1〕 **アドレスマッピング**　仮想メモリ機構には，実アドレスと仮想アドレスのマッピング（対応付け）機能が必要となる．

ファイル装置はメインメモリのバックアップメモリであるので，メインメモリに格納されているプログラムやデータは，原則として（一時的あるいは実行時データなどは除くと），ファイル装置にも存在する．一方，メインメモリ容量はファイル装置容量に比べて格段に小さいので，ファイル装置に格納されているプログラムやデータの一部だけが実行時（必要時）に一時的にメインメモリにコピーされることになる．

したがって，実メモリにアクセス対象が存在しない場合には，バックアップ

* フラグメンテーション（fragmentation）：メモリ領域の割付け/解放を繰り返すと未使用領域が徐々に小片化/断片化し，割付け可能な領域が不連続になること．「外部フラグメンテーション」ともいう．これに対して，割付け領域サイズと実際に使用している領域サイズとの差によって生じる未使用領域を「内部フラグメンテーション」という．

メモリからそれをコピーする必要があり，実メモリのメインメモリにはアクセス頻度の高い（参照局所性が高い，6.1.1項〔5〕参照）プログラムやデータを置いておくことが望ましい．

仮想アドレス空間のうちアクセス頻度が高い一部の空間が実アドレス空間にマッピングされる．

〔2〕　**動的アドレス変換機構(DAT)**　仮想メモリを利用する場合には，マシン命令のメモリオペランドは仮想アドレスであり，この場合には，アドレス変換は必ず行わなければならない．したがって，アドレス変換は動的機能で高速性が要件となるので，この機能のほとんどはハードウェア機構によって実現する．

仮想アドレスから実アドレスへの変換（**アドレス変換**（address translation）という）は実行時に行う．これを実現する機構を **DAT**(dynamic address translator：動的アドレス変換機構）という．

DAT はマシン命令における仮想アドレスをメインメモリの実アドレスに変換する．したがって，DAT には，仮想アドレスと実アドレスのマッピングを行うアドレス変換テーブル（**マッピングテーブル**）が必要となる．マッピングテーブルはマッピングの自由度やテーブルサイズなどを考えて，通常，メインメモリ上に置かれる．また，DAT はプロセッサの命令デコード機構（デコーダ）とメインメモリとの間に置かれ，前段には，アドレス修飾（2.2.3項〔5〕参照）機構がある．この場合は，まずアドレス修飾に従って実効アドレス（仮想アドレス）が求められ，それを DAT によって実アドレスに変換する．

〔3〕　**DAT によるアドレス変換の手順**　DAT を介しての実アドレス空間へのアクセスは次のような手順で実行される．

(1)　(必要なら)アドレス修飾によって実効アドレスを仮想アドレスとして生成する．

(2)「アクセスの要求された仮想アドレスが実アドレス空間にマッピングされているかいないか？」をチェックする．

①　あれば，マッピングテーブルによって仮想アドレスを実アドレスに変換（アドレス変換）する．

②　なければ，割込み（**フォールト**（fault）という）を発生し，必要な仮想アドレス空間を実アドレス空間の一部と置換（**ブロック置換**という，

6.2.4 項で詳述）する．それから，①を行う．

(3) 変換された実アドレスによってメインメモリ（ハードウェア）にアクセスする．

6.2.3 アドレス変換

〔1〕 **マッピング単位** 実メモリと仮想メモリ間でのアドレスマッピングはブロック（block：まとまりのある単位）ごとに行われる．このマッピングブロックの単位によってマッピング方式を次の2つに大別できる．

(a) **ページング**（paging）：**ページ**（page）という，ある決まった（固定長，通常は256～ 4Kバイト）サイズのブロックをマッピング単位とする．ページサイズは仮想メモリ機構の設計時に決められ，プログラムブロックやデータブロックの論理的な意味やサイズは考慮されずに，メモリ空間を機械的にページサイズで分割する．（〔2〕で詳述）

(b) **セグメンテーション**（segmentation）：論理的な意味を持つまとまりのプログラムブロックとかデータブロックを**セグメント**（segment）といい，このセグメントをマッピング単位とする．セグメントはプログラムやデータの論理的な意味やサイズを保つように設定されているので，セグメントサイズは固定長ではなく可変長である．（〔3〕で詳述）

物理的な均質性（マッピング単位が固定長）を活用するページングと論理的な不均質性（マッピング単位が論理的な意味を持つ）を重視するセグメンテーションは，1.1.3 項〔4〕で述べたように，トレードオフとなる．

〔2〕 **ページング** 図 6.5 に示すように，固定長のページ単位で実アドレス空間（物理ページ）と仮想アドレス空間（論理ページ）とのマッピングをとる．ページングで用いるアドレス変換テーブルを**ページテーブル**という．

仮想アドレス V は仮想ページ番号 v_p とページ内相対アドレス（オフセット）o で表す．ページングにおけるアドレス変換の手順は次のようになる．（図 6.5 参照）

(1) 仮想ページ番号 v_p でページテーブルを検索して実ページ番号 r_p を得る．

(2) 実ページ番号（実ページの開始アドレス）とオフセットによって実アドレス R を求める．

マッピング方式である．k 個のキャッシュブロックを1セットとするセットアソシアティブマッピングを「k ウェイ（k-way）セットアソシアティブマッピング」という．

図6.17に示すように，キャッシュのセット数 N_C は

$$N_C = \frac{B_C}{k} \tag{6.11}$$

となる．セットを決める第1段階のダイレクトマッピングは N_C セットに対して行い，第2段階のセット内の連想検索（フルアソシアティブマッピング）はマッピングテーブル（連想メモリ）の k 個のセル（キャッシュブロック）に対して行う．

セットアソシアティブマッピングのキャッシュへの（ヒット時の）アクセス手順は次のようになる．

(1) セットアドレス（「セット番号」あるいは「マッピングテーブル番号」ともいう）S によってマッピングテーブルの1セットを選択する．

(2) そのマッピングテーブルの1セット内の k セルを並列検索（連想）し，アクセス対象のメインメモリブロックのコピーがキャッシュ内に存在するかどうかをチェックし，存在すれば（ヒット時）そのブロックアドレス（キャッシュ内ブロックアドレス）を得る．

(3) ブロック内（相対）アドレスによって対象アドレスにアクセスする．

セットアソシアティブ方式においては

$$1 \leqq k \leqq B_C \tag{6.12}$$

であり，式（6.11）より，k によって

$$1 \leqq N_C \leqq B_C \tag{6.13}$$

の範囲で N_C が決まる．すなわち

- ダイレクトマッピングは，$k=1$（ウェイ），$N_C=B_C$（セット）の「1ウェイセットアソシアティブマッピング」
- フルアソシアティブマッピングは，$k=B_C$（ウェイ），$N_C=1$（セット）の「B_C ウェイセットアソシアティブマッピング」

にそれぞれ当たる．

セットアソシアティブ方式では，k セルの連想メモリを N_C 個必要とし，連想メモリ全体の総セル数は B_C で，フルアソシアティブと同じである．しかし，1個の

コヒーレンシが保たれていない状態（**ダーティ**（dirty）という）期間がある．したがって，コヒーレンシを管理/修復（保持）するハードウェア機構が必要となり，また，その制御は複雑である．

③ ブロックサイズが大きいとブロック単位での書戻し時に更新されていないデータも書き戻すむだが多くなり，書戻し時間の増大を招く．

①の長所によって得られるキャッシュ性能を重視する場合には，このライトバック方式がとられる．

ライトバックでは，キャッシュへの書込みアクセスがあったかどうかを示すフラグとして**更新ビット**が必要である．更新ビットはマッピングテーブル（タグ）に付加される．

〔3〕 **書込みアクセスのミスヒット時の処理** データキャッシュへのアクセスは，読出しだけではなく書込みもある．通常は，「直前に読み出されたデータそのものへの上書きかその近接アドレスへの書込みが多い（参照局所性がある）」という理由で，書込みアクセスのヒット率は読出しアクセスよりも高いといわれている．

しかし，データキャッシュへの書込みアクセスがミスヒットする場合もある．その場合，ライトスルーでは，次に示すいずれかの方法によってメインメモリを更新する．（図 6.20 参照）

(a) **ノーライトアロケート**（no write allocate）：メインメモリへの書込みだ

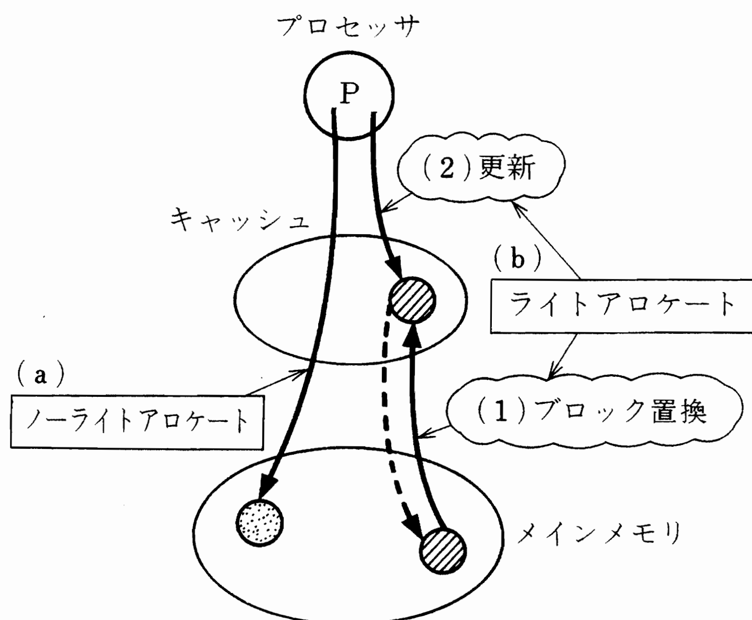

図 6.20 書込みアクセスのミスヒット時のメインメモリ更新方式

として想定している．この場合には，前半期（15回）で本書の第1～4章を，後半期（15回）で第5，6章を，それぞれ講述・学習するとよい．第7章は実際のアーキテクチャ例や最近のトピックスを知るために選択的に講述・学習するか，大学院科目内容に回すとよい．もし，本講義科目に15回（半年）しか割り当てられないならば，第2，5，6章を中心に講述・学習し，第1，3，4，7章は選択的に講述・補習するとよい．

各章末の「演習課題」はオープンクエスチョン（公開質問状）であり，著者自身まだ解答を持っていないものを挙げた．これらの課題にコメントすることが「これからのコンピュータアーキテクチャ学の取り組むべき課題」と考えている．どうか，1題でも良いから考察を加え，そのレポートを著者宛（電子メールアドレス：shibayam@kit.jp）に送ってほしい．

現代のコンピュータメーカの世界は（やむを得ずかもしれないが）「性能第一主義」に陥っている．しかし，現代のコンピュータ技術を支えているのは「芸術」に近いユニークさやオリジナリティであることを思い出してほしい．今では商用マイクロプロセッサの性能競争の最先端に位置するRISCもスーパスカラアーキテクチャも，既製品や既成事実をひっくり返すことから発達してきた．本書の読者諸氏や演習課題のレポート提出者から，「世の中に1台しかないコンピュータアーキテクチャ」の設計者が生まれることを望みたい．

本学の同僚である平田博章氏には，丁寧な査読をして頂き，山のような誤りの指摘やコメントを頂いた．ここに深謝いたします．また，コンピュータサイエンスの専門コア科目として必要最小限の事項を1冊に詰めたために，本書は，教科書としては，我が国の出版界の常識から少々はずれた大部である．それが日の目を見るのは，オーム社出版部の英断である．ここに感謝します．

最後に，「ハードウェアとソフトウェアのいずれが欠けても良いアーキテクチャはできない」ことを実践的に教えてくれる我が家のアーキテクト：妻・真木子と4人の子供達・風野，すみれ，ののみ，蒼宙に，心から「おおきに」．

なお，著者のプロフィールは，公私ともども，

http://www.ark.is.kit.ac.jp/~shibayam/

にある．どうぞ時間があれば立ち寄って下さい．

1997年早春　京都・松ケ崎にて

柴　山　　潔